Intermediate Programming

Lecture #9 – Exception Handling

Runtime Errors

- Sometimes programs are stopped because their errors are so severe that they cannot recover from them. These are called fatal errors (or unrecoverable errors). They are also called exceptions, and some require the attention of the operating system or the programmer.
- They are also called runtime errors because they occur when the program is running.

Anticipating Runtime Errors

- It is almost always in the programmer's best interest to catch these errors himself (or herself) because the programmer can usually handle it more gracefully than the operating system can.
- Java provides a mechanism for handling both standard exceptions (errors anticipated by the Java compiler) and non-standard exceptions.

BadDivision.java

```
// A sample of some bad programming
public class BadDivision  {
  public static void main(String[] args) {
    int x, y, z;
    x = 5;
    y = 0;

    z = x / y;
    System.out.println("The answer is " +  z);
  }
}
```

Output for `BadDivision.java`

```
ArithmeticException: / by zero
  at BadDivision.main(BadDivision.java:8)
  at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
  at sun.reflect.NativeMethodAccessorImpl.invoke(Unknown Source)
  at sun.reflect.DelegatingMethodAccessorImpl.invoke(Unknown Source)
  at java.lang.reflect.Method.invoke(Unknown Source)
```

What is an exception?

- To Java, an exception is an object that has a base class called Exception. It is possible to create ones own classes of exceptions and use them to handle errors in a graceful, proper fashion.

`try` and `catch`

- It is problematic to have the operating system try to deal with exceptions, especially the ones that we should be able to anticipate.
- We can have the methods deal with their own problems by using try and catch.
- The exception that occurs in a try block is handled by the appropriate catch block.

Syntax for `try` and `catch`

```
try  {
  Some statements where an exception may
  occur
}  catch (AnExceptionType exception) {
  Some statement to handle the exception
}
```

BadDivision.java
Rewritten to Handle the Exception

```
// A sample of some bad programming
public class BadDivision  {
  public static void main(String[] args) {
    int x, y, z;
    x = 5;
    y = 0;

    try  {
      z = x / y;
    }
    catch (ArithmeticException e) {
      System.out.println("Oops! " + e.getMessage());
      z = 0; // Necessary so that we can print z
    }
    System.out.println("The answer is " +  z);
  }
}
```

Can We Design Our Own Exceptions?

- Yes. We can do this using instances of the base class Exception.
- We can create derived classes of the base class exceptions.
- We can use one or more of the library exception classes that are part of Java.

Example: **BadIO.java**

```
import java.util.*;

// An illustration of another type of exception
public class BadIO {
  public static void main(String[] args) {
    int x, y, z;

    x = readInput();
    System.out.println("The input is " + x);
  }
```

```
  // readInput() - Reads an integer input
  public static int readInput() {
    Scanner keyb = new Scanner(System.in);
    try {
      // Get the input (which should be an integer)
      System.out.println("Enter an integer\t?");
      return (keyb.nextInt());
    }
    catch (InputMismatchException e) {
      // Maybe the input isn't an integer
      System.out.println(e.getMessage());
      System.out.println("Oops!");
      return 4;
    }
  }
}
```

Another Example: **DanceLesson.java**

```
import java.util.Scanner;

public class DanceLesson {
  public static void main(String[] args) {
    Scanner keyb = new Scanner(System.in);

    System.out.println
           ("How many male dancers are there?");
    int men = keyb.nextInt();

    System.out.println
           ("How many female dancers are there?");
    int women = keyb.nextInt();
```

```
    if (men == 0 && women == 0) {
      System.out.println
           ("Lesson is canceled.  No students.");
      System.exit(0);
    }
    else if (men == 0) {
      System.out.println
             ("Lesson is canceled.  No men.");
      System.exit(0);
    }
    else if (women == 0) {
      System.out.println
             ("Lesson is canceled.  No women.");
      System.exit(0);
    }
```

```
    // women >= 0 && men >= 0
    if (women >= men)
      System.out.println
        ("Each man must dance with "
            + women/(double)men + " women");
    else
      System.out.println
        ("Each woman must dance with "
                    + men/(double)women + " men");
    System.out.println("Begin the lesson.");
  }
}
```

DanceLesson2.java with **try** and **catch**

```
import java.util.Scanner;

public class DanceLesson2 {
  public static void main(String[] args) {
    Scanner keyb = new Scanner(System.in);

    System.out.println
         ("How many male dancers are there?");
    int men = keyb.nextInt();

    System.out.println
         ("How many female dancers are there?");
    int women = keyb.nextInt();
```

```
// This may illustrate the syntax but it isn't a
//  good use of exception handling
// Here in the try block, we place the code that
// is likely to cause the exception
try  {
  if (men == 0 && women == 0)
    throw new Exception
         ("Lesson is canceled.  No students.");
  else if (men == 0)
    throw new Exception
         ("Lesson is canceled.  No men.");
  else if (women == 0)
    throw new Exception
         ("Lesson is canceled.  No women.");
```

```
  // women >= 0 && men >= 0
  if (women >= men)
    System.out.println
         ("Each man must dance with "
              + women/(double)men + " women");
  else
    System.out.println
       ("Each woman must dance with "
                + men/(double)women + " men");
}
```

```
    // Here in the catch block, we place the code
    // that we will execute if there is an exception
    // found.
    // If we don't find an exception we skip down
    // the next block of code.
    catch (Exception e) {
      String message = e.getMessage();
      System.out.println(message);
      System.exit(0);
    }

    System.out.println("Begin the lesson.");
  }
}
```

`throw` Statement

- The syntax for a throw statement is:

  `throw new` *ExceptionClassName*
  `(`*MaybeSomeArguments***`) ;`**

- When the throw statement is executed, the execution of the surrounding try block is stopped and (normally) control is transferred to a catch block. The code in the catch block is executed next.
- Example

```
throw new Exception("Division By Zero");
```

`catch` Block Syntax

- The syntax for a `catch` block is:

  `catch (`*ExceptionClass CatchBlockParam*`) {`**

  Code to be performed if an exception with this name is thrown in the try block.

  }
- The catch block parameter is an identifier in the heading of a catch block that serves as a placeholder for an exception that might be thrown. When a suitable exception is thrown in the preceding try block, that exception is plugged in the for the catch block parameter. While e is typically used, it can be any identifier.
- Example

```
catch (Exception e) {
    System.out.println(e.getMessage());
    System.exit(0);
}
```

`getMessage()`

- Every exception has a String instance variable that contains some message, which typically identifies the reason for the exception.
- **`e.getMessage()`** returns the message contained within the exception object **`e`**.
- Example

```
throw new Exception("Input must be positive.");
```

is caught by

```
Catch (Exception e) {
  System.out.println(e.getMessage());
  System.exit(0);
}
```

DivisionByZeroException.java

```
// A derived class of exceptions
public class DivisionByZeroException
                                extends Exception {
  // While a constructor could have more,
  // this is a common form.
  public DivisionByZeroException() {
    super("Division by zero!!");
  }

  // We're invoking the base class's constructor
  public DivisionByZeroException(String message) {
    super(message);
  }
}
```

DivisionDemo.java

```
import java.util.Scanner;

public class DivisionDemo  {
  public static void main(String[] args)  {
    try {
      Scanner keyb = new Scanner(System.in);

      System.out.println("Enter numerator:");
      int numerator = keyb.nextInt();
      System.out.println("Enter denominator:");
      int denominator = keyb.nextInt();

      if (denominator == 0)
        throw new DivisionByZeroException();
```

```
    double quotient
         = numerator/(double) denominator;
    System.out.println(numerator + "/"
             + denominator + "=" + quotient);
  }
  catch(DivisionByZeroException e) {
    System.out.println(e.getMessage());
    secondChance();
  }

  System.out.println("End of program.");
}
```

```
public static void secondChance() {
  Scanner keyb = new Scanner(System.in);

  System.out.println("Try again:");
  System.out.println("Enter numerator:");
  int numerator = keyb.nextInt();
  System.out.println("Enter denominator:");
  int denominator = keyb.nextInt();

  if (denominator == 0)  {
    System.out.println("I cannot divide by 0");
    System.out.println("Aborting program.");
    System.exit(0);
  }
```

```
    double quotient
          = ((double) numerator)/denominator;
    System.out.println(numerator + "/" + denominator
                       + "=" + quotient);
  }
}
```

`Exception` Object Characteristics

- The two most important things about an exception object are its type (the exception class or a subclass) and a message that it carries in an instance variable (that is usually a String).

Defining an `Exception` Subclass

- If you have no compelling reason to use any other class as the base class, use **`Exception`** as a base class.
- You should define two (or more) constructors (more on this below).
- Your exception class inherits the method **`getMessage()`**. Normally you do not need to add any other method, but it's legal to do so.

Defining an `Exception` Subclass (continued)

- You should start each constructor definition with a call to the constructor of the base class, such as the following:

```
super("Sample Exception thrown.");
```

- You should include a nor-argument constructor, in which case, the call to **`super`** should have a string argument that indicates what kind of exception it is. This string can then be recovered by using the **`getMessage()`** method.
- You should also include a constructor that takes a single string argument. In this case, the string should be an argument in a call to **`super`**. That way, the string can be recovered with a call to **`getMessage()`**.

SampleException.java

```
public class SampleException extends Exception {
  public SampleException() {
    super("Sample exception thrown!");
  }

  public SampleException(String message) {
    super(message);
  }
}
```

Exception's Message and Type

- The data type of the message being passed by the Exception does not have to be a String.
- Under some circumstances, there may be very good reason to let it be a string.

BadNumberException.java

```
public class BadNumberException extends Exception  {
  private int badNumber;

  // BadNumberException() - This constructor sets
  //                        the number
  public BadNumberException(int number)  {
    super("BadNumberException");
    badNumber = number;
  }

  // BadNumberException() - A default constructor
  public BadNumberException()  {
    super("BadNumberException");
  }
```

```
  // BadNumberException() - This constructor sets
  //                        the message
  public BadNumberException(String message)  {
    super(message);
  }

  // getBadNumber() - An accessor for the bad number
  public int getBadNumber() {
    return badNumber;
  }
}
```

BadNumberExceptionDemo.java

```
import java.util.Scanner;

public class BadNumberExceptionDemo  {
  public static void main(String[] args) {
    try {
      Scanner keyb = new Scanner(System.in);

      System.out.println
           ("Enter one of the number 42 and 24:");
      int inputNumber = keyb.nextInt();

      if ((inputNumber != 42)
                    && (inputNumber != 24))
        throw new BadNumberException(inputNumber);
```

```
      System.out.println
         ("Thank you for entering " + inputNumber);
    }

    catch (BadNumberException e) {
      System.out.println(e.getBadNumber()
                    + " is not what I asked for.");
    }
    System.out.println("End of program.");
  }
}
```

Working With Multiple Exceptions

- It is possible to handle the possibilities of more than one exception.
- The try block throws the exceptions that it needs to throw.
- Then it is a matter of having multiple catch blocks, one per exception type.

MoreCatchBlocksDemo.java

```
import java.util.Scanner;

public class MoreCatchBlocksDemo  {
  public static void main(String[] args) {
    Scanner keyb = new Scanner(System.in);

    try {
      System.out.println
              ("How many pencils do you have?");
      int pencils = keyb.nextInt();

      if (pencils < 0)
        throw new
              NegativeNumberException("pencils");

      System.out.println
                ("How many eraser do you have");
      int erasers = keyb.nextInt();
```

```
    double pencilsPerEraser;

    if (erasers < 0)
      throw new
            NegativeNumberException("erasers");
    else if (erasers != 0)
      pencilsPerEraser = pencils/(double) erasers;
    else
      throw new DivisionByZeroException();

    System.out.println
       ("Each erase must last through "
                + pencilsPerEraser + " pencils.");
  }
```

```
    catch (NegativeNumberException e) {
      System.out.println
           ("Cannot have a negative number of "
               + e.getMessage());
    }
    catch (DivisionByZeroException e) {
      System.out.println("Do not make any
  mistakes.");
    }

    System.out.println("End of program.");
  }
}
```

NegativeNumberException.java

```
public class NegativeNumberException
                              extends Exception  {
  // BadNumberException() - A default constructor
  public NegativeNumberException()  {
    super("NegativeNumberException");
  }

  // NegativeNumberException() - This constructor
  //                             sets the message
  public NegativeNumberException(String message)  {
    super(message);
  }
}
```

Throwing Exceptions in a Method

- Sometimes it may make sense to throw an exception in a method, even though you have no desire to catch it.
- In cases, like these, you can declare the exception in the method header:

```
public void sampleMethod() throws SampleException {
```

- If there is more than one exception to be thrown, they are listed in the header, separated by commas:

```
public void sampleMethod()
        throws SampleException, OtherSampleException {
```

DivisionDemo2.java

```
import java.util.Scanner;

public class DivisionDemo2  {
  public static void main(String[] args)  {
    Scanner keyb = new Scanner(System.in);

    try {
      System.out.println("Enter numerator:");
      int numerator = keyb.nextInt();
      System.out.println("Enter denominator:");
      int denominator = keyb.nextInt();

      double quotient
            = safeDivide(numerator, denominator);
      System.out.println
           (numerator + "/" + denominator
                           + "=" + quotient);
    }
```

```
    catch(DivisionByZeroException e) {
      System.out.println(e.getMessage());
      secondChance();
    }

    System.out.println("End of program.");
  }

  public static double safeDivide
             (int top, int bottom)
                   throws DivisionByZeroException {
    if (bottom == 0)
      throw new DivisionByZeroException();
    return top/(double)bottom;
  }
```

```
  public static void secondChance() {
    Scanner keyb = new Scanner(System.in);

    try {
      System.out.println("Enter numerator:");
      int numerator = keyb.nextInt();
      System.out.println("Enter denominator:");
      int denominator = keyb.nextInt();

      double quotient = safeDivide
                    (numerator, denominator);
      System.out.println(numerator + "/"
           + denominator + "=" + quotient);
    }
```

```
    catch (DivisionByZeroException e) {
      System.out.println
                ("I cannot do division by zero.");
      System.out.println("Aborting program.");
      System.exit(0);
    }
  }
}
```